Issues in Multiprocessor Memory Consistency Protocol Design and Verification

Ritwik Bhattacharya and Ganesh Gopalakrishnan *
School of Computing, University of Utah
{ritwik | ganesh}@cs.utah.edu

July 4, 2002

Abstract

Distributed shared memory (DSM) systems are central to our advancement in terms of high-performance computing. The complexity of DSM protocol design stems both from the complexity of the high-level notion of correctness—conformance to a modern weak shared memory model—and the degree to which these protocols are aggressive. Many similar issues are faced in the design of related protocols such as in input/output subsystems. In this paper we provide a glimpse of the complexity faced, and our work in progress in addressing these issues through model-checking and synthesis using recently proposed challenge problems to drive our research.

1 Introduction

The computer industry has consistently delivered increasing performance with each new generation of processors. To sustain the current level of growth, *system design and verification complexity* must be kept in check. Design and verification complexity can be minimized through *modular design principles* that allow design and verification results to be *reused*, and through *provably correct automation* that allows verification to be done at *higher level*s. The problem addressed in this paper is how to significantly elevate the level of modularity and automation in the design of *distributed shared memory* (DSM) systems, where the verification complexity has grown out of proportion with raw transistor count.

The DSM discipline continues to be a dominant organizational paradigm for computers. DSM machines may be as simple as a dual-processor desktop computer or as sophisticated as the 512-node ASCI White [1] of Lawrence Livermore. The inherent verification complexity of these DSM machines will be exacerbated by the fact that they will be integrated on single chips. Examples of chip level multiprocessors include the IBM Power4 [2], the Compaq Piranha [10], and the SUN microsystems MAJC 5200 [3]. These chips consist of ensembles of processors, instruction and data level 1 (L1) caches, L2 caches, memory

*This work was supported by National Science Foundation Grants CCR-9987516 and CCR-0081406

controllers, packet switchers, DSM protocol engines, and router hubs. For example, the Piranha has all this functionality available within a *single* 100+ million transistor die. All these processors face a common set of issues when it comes to their DSM protocol design:

- Given the growing computation to communication delays, ownership migrations and invalidations must be handled inexpensively.
- Given the finiteness of resources such as buffer locations, transaction identifiers, etc, deadlock avoidance and buffer reservation schemes must be built into protocols.
- Given the ever widening gap between CPU speeds and memory system speeds, read latencies must be hidden at every possible opportunity. In particular, as many operations must be allowed to happen out of order, implying that *weak memory consistency models* [8] be employed.

The above mentioned problems pertaining to DSM protocol design are, to a large measure, shared by emerging I/O system standards such as 3GIO [4], Infiniband [38], as well as protocols used in networked embedded systems such as Bluetooth [18]. While many implementation methods provide the ability to correct mistakes late in the design cycle (software implementations of DSM protocol engines [11, 23], and software [24] or microcode [31, 10] implementations of hardware DSM protocol engines, for example), the extent of error recovery possible is limited, and the debugging costs are high.

In this paper, we report on our research in progress at the University of Utah which is addressing some of the above issues. The overall goals of our research are as follows. First, we aim to develop techniques that apply to real industrial-scale protocols. Second, we aim to capture recurring design situations as "idioms" and develop reusable verification or synthesis techniques. Last but not least, we aim to have "brute force" verification techniques available to designers so that they can deploy them on new designs that break past patterns, while a formal understanding is being obtained for them. In this paper, we present our specific efforts to achieve the above goals:

- We briefly summarize some of the issues that make DSM protocols complex. To shed more light on these issues, we examine the Wildfire [26] protocol in some detail.
- We survey some of our past efforts in capturing design patterns and setting up a derivational-style synthesis process. We provide a preliminary evaluation of the success of this approach on the Wildfire protocol. This preliminary evaluation reveals how far we can go with respect to an industrial-scale protocol, and what remains to be done.
- We summarize our efforts so far in using "brute force" model-checking, and present our plans to make this process more efficient.

In the remainder of this section, we review some basic terminology and briefly survey related work. In Section 2, we summarize some of the general difficulties of DSM protocol design, and specific issues that come up in the Wildfire protocol. In Section 3, we present the approaches we have tried in the past, as well as plan to try in the near future. Section 4 concludes the paper.

Basic Terminology

DSM protocols have one purpose: manage the *ownership* of cache lines so that maximal concurrency of access is permitted, and the values returned by the reads are according to the desired shared memory model [8]. To understand

shared memory models, consider a simple example. In a multiprocessor, a program `write(A,1); read(B)` running on processor P1 and another program `write(B,2); read(A)` running on processor P2 interleave, producing different execution results for the `read` depending on how *weak* the memory model is: under a strong model such as sequential consistency, one of the `read`s must return[1]either `1` or `2`, while in a weak model such as Total Store Ordering [39], both the reads can return `0`. To drive home the connection between memory models and synchronization code, consider one simple example, namely Peterson's algorithm for mutual exclusion [35]. This mutual exclusion protocol fails to work under TSO, but works under sequential consistency.

In general, modern weak shared memory models are far more intricate than either sequential consistency or cache coherence. They support ordinary- as well as special reads and writes that obey different ordering properties. Furthermore, these orderings depend on which address locations - coherent, non-coherent, or I/O - these reads and writes fall on. While the weak ordering rules impose a burden on writers of synchronization libraries, they provide considerably more opportunities for compiler writers and hardware designers to gain performance through re-orderings.

The *correctness problem* that we allude to in this paper is one of showing that all executions generated by a DSM multiprocessor for any given concurrent program are also allowed by the shared memory consistency model. This goes far beyond the scope of what is traditionally known as "*cache coherence verification*" where the correctness is only with respect to a *single* address appearing coherent (consistent) for all processors. In this sense, both sequential consistency and TSO obey *coherence*; however, as the Peterson protocol example shows, they are not equivalent shared memory consistency protocols.

Related Work

The area of shared memory consistency models is vast. We do not attempt a survey; for details, please see [8, 5]. In [19], Grahn studied many contemporary DSM protocols in a unified setting. He also compared the implementation details of achieving synchronization. The use of formal operational models of memory consistency to verify assembly code synchronization routines is studied in [16]. In [34], several protocols, including Stanford FLASH [24, 20] are shown to be sequentially consistent, using theorem proving. Additional related works in specification and verification of DSM protocols include [29, 5, 14, 36, 7, 9].

2 Complexity of DSM protocols

2.1 General issues contributing to complexity

To provide a concrete context for our discussions, we now take-up one commonly occurring design scenario from DSM protocol design: the 'three-way handoff' scenario of handling a missed write at a processor, say P1. In this scenario, processor P1 requests the directory controller of a cache line for an exclusive copy of the cache line before it can proceed with the write. The directory controller, in turn, requests the current owner (another processor, say P2) of the line to forward the line directly to P1. Completely unawares, P2 may have already decided to evict the line, which is in flight in the form of a *line relinquish*

[1] We assume an initial memory value of `0` at all locations.

message towards the directory. In this case, the directory controller receives an "unexpected" line relinquish from P2 when it was expecting a "forwarding to P1 done" acknowledgement. In most designs, the directory controller would recover from such a state by giving priority to the cache controller's attempt over its own attempt. Consequently the directory controller acts as if it has been *implicitly nacked*[2]. It collects P2's relinquished line and hands it over to P1. Over and above these correctness considerations, one must also take into account the other crucial factors, such as the following:

- *Message ordering properties of the interconnect medium:* usually several *priority lanes* are employed to allow messages to re-order, partly to improve performance, and partly to avoid deadlocks. Each message above must be sent on the 'correct' priority lane.
- *Buffer capacities:* Usually, before sending a request, a requester, X, must reserve a spare location in its input queue to be able to receive an acknowledgement from its requestee, Y. However, sometimes a single spare location may not suffice, as the following scenario of 'tail of buffer livelock' illustrates. In this scenario, when the acknowledgement is outstanding, another requester, Z, may send a request to X. This may happen precisely when Y is about to respond. Since the only available buffer slot is occupied by Z's request, X is forced to send back Y's acknowledgement. It then examines Z's request which, it usually cannot process, as X is "in the middle of a request itself." X, thereby, ends up *nack*ing Z's request also. This scenario can repeat indefinitely.
- *Respect the memory ordering semantics:* The above protocol actions describe how *coherence* - strong ordering with respect to a *single* cache-line - is attempted to be maintained. It is unclear whether the ordering constraints for *multiple addresses* as dictated by *sequential consistency* or *weaker memory models* are being met.

The Wildfire protocol involves virtually all the above issues plus some, as described in the next section.

2.2 Additional complexities of the Wildfire protocol

We first describe briefly the Alpha memory model and the Wildfire cache coherence protocol, which we are using as a driving example in our research. We then give some example scenarios that show the complexity of the protocol.

2.2.1 The Alpha Memory Model

In the Alpha memory model we consider [26], a processor can issue five different kinds of memory requests, which are named Rd, Wr, LL, SC and MB. A Rd is a read of a memory address, and the memory responds with a data value. A Wr is a write to a memory address. The memory responds with an acknowledgement that the write has been performed. An LL is a 'Load Locked.' This is a read of a memory address that 'locks' the address for the processor, unlocking any other address that might be locked for that processor. The memory responds with a data value. An SC is a 'Store Conditional.' This is a conditional write of a memory address which succeeds if the specified address is locked, and fails otherwise. If it fails, it does not update memory. An SC always (successful or

[2] *nack* stands for negative acknowledgement.

not) unlocks any address that is locked by the processor. A memory address is also unlocked when a different processor performs a Wr or a successful SC to an address locked by the processor. The memory responds to an SC with an acknowledgement indicating whether the request succeeded or not. An MB is a 'Memory Barrier.' Two requests from the same processor to the same address must always be performed in the order in which they were requested. Two requests to different addresses from the same processor must be performed in the order they were requested if an MB request was issued after the first and before the second. The memory does not respond to an MB. The *source* of a memory operation is the most recent write to the memory address involved in the operation (see ordering rules below).

The orderings that the Alpha memory model imposes on the above five operations are as follows:

- A request r_1 precedes another request r_2 if r_1 is the *source* of r_2.
- r_1 precedes r_2 if r_1 and r_2 are requests from the same processor, r_1 precedes r_2 in the request queue, and either at least one of them is an MB, or they are requests to the same address.
- Writes and successful SCs to the same location that have issued responses are totally ordered.
- LL's and successful SC's are properly paired, without intervening writes by another processor to the same address.

2.2.2 The Wildfire Protocol

The Wildfire cache coherence protocol was designed to implement the Alpha memory model. It models a NUMA shared memory system, and consists of a network of processors and memory connected to local switches, which are all in turn connected to a single global switch. An example is shown in Fig 1. The global space of memory addresses is partitioned among the local switches, with each address belonging to exactly one local switch, which is the home node for that address. Each processor has a cache that contains local copies of some addresses. Wildfire is a directory based cache coherence protocol where each local switch maintains a directory with entries for each address that belongs to it. Each entry has information on which processors currently have a copy of that address in their local cache.

Processors communicate with each other and with local switches through messages. If a processor needs to send a message to a local switch other than the one it is attached to, it sends the message to its local switch, which relays it to the global switch, which in turn relays it to the correct local switch.

Processors are connected to their local switch by two unidirectional queues. The same is true for local switches and the global switch. Messages in the queues are allowed to reorder, subject to certain constraints. All request and control messages travel along this set of queues. There is also a separate network of queues that connect processors directly to one another. These queues are used only to send actual data values by the owner of an address.

An entry in the processor's cache is in one of the following states :

- *Exclusive*: This is the primary copy of the data. The processor can read or write it.
- *SharedClean*: The copy is a secondary, read-only copy.
- *SharedDirty*: This is the primary copy of the data, but it is read-only (The state was Exclusive, and then some other processor requested a read-only copy).

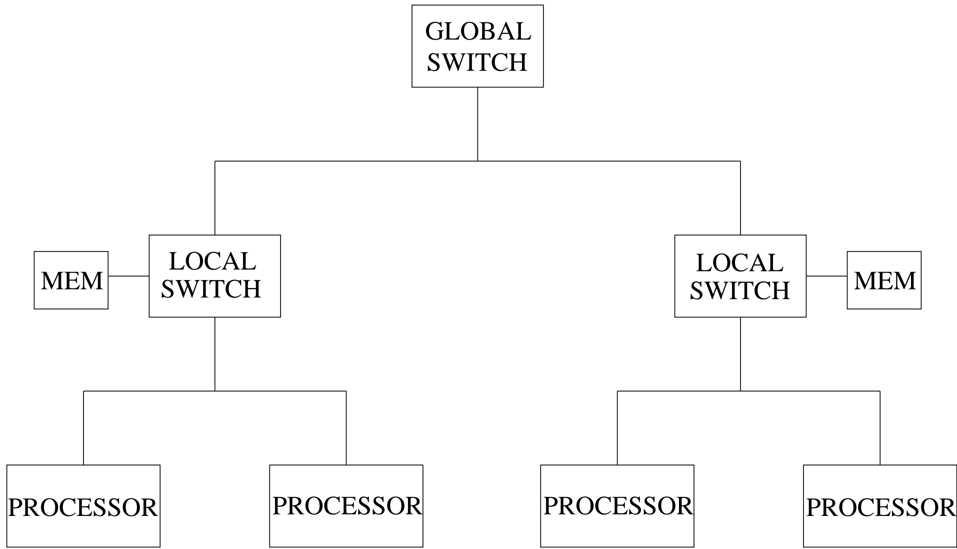

Figure 1: Example of a Wildfire multiprocessor system

- *Invalid*: The cache does not contain a copy of this address.

The state of a cache line changes as a result of the various requests that processors issue, and the responses to those requests. As mentioned earlier, these communications are carried out through messages. An indication of the complexity of the protocol is that there are 13 different *types* of messages in the protocol, and there can of course be multiple messages of each type at any given time in the system. Completing a single memory operation (such as a Wr or a Rd) can require exchanging as many as seven messages among three different components of the system, all of which can be interleaved and reordered with other messages in the system. There are many other subtleties about the protocol that lead to its complexity, but are beyond the scope of this paper. Please see the full Wildfire documentation at [26] for more information.

2.2.3 Scenarios of the Wildfire protocol

We now discuss some scenarios that illustrate the complexity of the Wildfire protocol. First, we describe how Wildfire handles the situation described in Section 2.1. A processor maintains multiple *versions* of the same cache line, each with a possibly different data value. Thus, when a processor evicts a cache line, it does not delete the version of the cache entry until an *ack* for the eviction has been received. So when its evict message crosses over with the request from the directory to forward the line, it can handle the forwarding request, since it has kept the version around. This data is then directly sent to the requesting processor via a separate 'fill channel.'

Our second scenario describes what is known as the shadow mode in the Wildfire protocol. For this scenario, suppose that processors p_1 and p_3 and address a_1 are on local switch ls_1, and processors p_2 and p_4, and address a_2 are on local switch ls_2. Also, assume that p_1 has the primary copy of a_2 in it's cache, and similarly p_2 has the primary copy of a_1. Now, the following events occur [26]:

– p_1 requests a copy of a_1.

- p_1's request reaches ls_1, which sends a request towards p_2 to forward the data to p_1, and updates it's directory entry for a_1 to record p_1 as the new owner.
- p_2 requests a copy of a_2.
- p_2's request reaches ls_2, which sends a request towards p_1 to forward the data to p_2, and updates it's directory entry for a_2 to record p_2 as the new owner.
- p_3 requests a copy of a_1.
- p_3's request reaches ls_1, which directly puts a request to forward the data in p_1's queue.
- p_4 requests a copy of a_2.
- p_4's request reaches ls_2, which directly puts a request to forward the data in p_2's queue.
- Lots of other unrelated messages enter the queues for p_1 and p_2
- The forward request for p_1 to send the line a_2 to p_2 reaches ls_1
- The forward request for p_2 to send the line a_1 to p_1 reaches ls_2

Now, the request for p_1 to forward the line a_1 to p_3 is at the head of p_1's queue, but it cannot handle it yet, since it does not have the line a_1 (though the directory ls_1 thinks it does). Similarly, the request for p_2 to forward the line a_2 to p_4 is at the head of its queue, but p_2 cannot handle it either, since it does not have the line a_2. The request for p_2 to forward the line a_1 to p_1 is stuck at ls_2, since p_2's queue is full with the other unrelated messages. Similarly, the request for p_1 to forward the line a_2 to p_2 is stuck at ls_1, since p_1's queue is full with the other unrelated messages too. We have now reached a deadlock, essentially due to the fact that p_3's (p_4's) request for line a_1 (a_2) gets directly inserted into p_1's (p_2's) queue.

This situation is avoided in Wildfire by the use of 'shadowing'. What this says is that whenever an address that is owned by a directory is in the cache of a processor not directly connected to that directory (local switch), *all* messages related to that address must go through the global switch, even if both the source and destination of a message are on the same local switch. In the above situation, this condition will result in p_3's (p_4's) request for line a_1 (a_2) to be routed through the global switch, ensuring that either p_3's request for a_1 will end up behind p_2's request for a_2 in p_1's queue, or p_4's request for a_2 will end up behind p_1's request for a_1 in p_2's queue, thus eliminating the deadlock.

It is nontrivial to prove that the above solution works in general, and actually involves proving that Wildfire satisfies the liveness property of the Alpha memory model.

Hopefully, the above scenarios will serve to convince the reader that today's memory consistency protocols are highly complex systems, and hence verifying their correctness is commensurately hard.

3 Verification Effort

In this section, we describe our efforts at verifying the Wildfire protocol. We discuss our attempts at using the TLC model-checker[25] (3.1), the Murφ system [15] (3.2), and an experimental synthesis technique [30] (3.3).

3.1 Verification using TLC

Both the Alpha memory model specification and the Wildfire protocol are written in TLA[27], which is a logic for specifying and reasoning about concurrent systems. Hence, our initial attempt to verify the Wildfire protocol focussed on using TLC, which is an explicit state enumeration based model checker for specifications written in TLA. We soon realized, however, that TLC would not be adequate, for two reasons. For technical reasons that are beyond the scope of this paper, TLC cannot handle specifications that are not *machine closed*, which is the case with the Alpha specification. A specification is said to be machine closed if it's liveness property does not introduce additional safety properties into the system [6]. The second, practical reason is that TLC is written in Java, making it extremely slow. The Wildfire model was allowed to run for seven days, and TLC had only explored around 15 million states, for a rate of around 1400 states/minute. This was unacceptable for any practical verification to be done. The TLC verification run did find a 'bug', but that turned out to be a bug in TLC itself, and not the protocol. This was reported to the authors of TLC, who have said that TLC will be fixed to remove this bug. This bug has to do with the way TLC handles conjunctions.

3.2 Verification using Murφ

We next decided to code up the Wildfire protocol in a different modeling language, so we could carry out the verification with another tool. We debated on whether to use a symbolic state representation based tool such as SMV [28], or an explicit state enumeration based tool like Murφ or Spin. SMV was ruled out as its input language is too low level for coding the protocol at the required level of abstraction. Also, based on the experience of Hu [22], it seemed likely that there would be a considerable blow-up in the BDD size. In the end, we decided to go with Murφ, as it has a reasonably expressive input language, and is also quite efficient. The Murφ coding effort took about a month. During this time [37] adapted an existing parallel version of Murφ to run using MPI libraries, so it could be run on a network testbed at the University of Utah which provides up to 172 networked workstations to run experiments on. The sequential version of Murφ soon consumed all the memory resources of the largest single machine at our disposal, which is an SGI with 4 GB of RAM. We then started a run of the parallel version on the network testbed. Each machine in the testbed has 512 MB of RAM. Our initial experiment with 15 such nodes explored about 3.3 million states in 4 minutes before crashing the system. This crash is thought to be due to a bug either in the MPI libraries, or the port of Murφ to MPI, and is being investigated. But the important lesson we learned is that even for the small instantiation of the problem we chose, with two processors, one address, and queue sizes of three, the theoretical total state space is around 48 million (using hashing of the state vector to 40 bits), suggesting that for a realistic instantiation, the state space might be beyond the capabilities of conventional model checking. This has lead us to explore a completely different technique for deriving such protocols, which we discuss in the next section.

3.3 DSM Protocol Synthesis

There have been very few attempts at developing formal approaches to DSM protocol design that take higher level protocol specifications and generate effi-

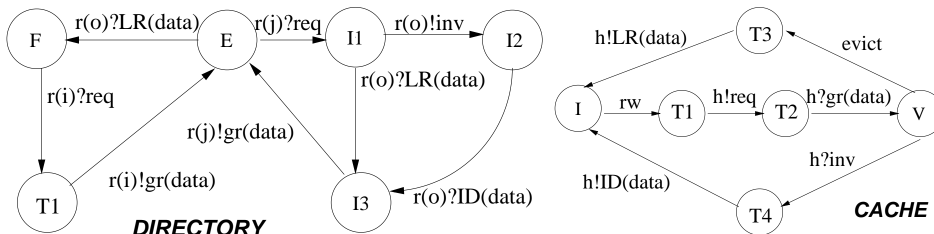

Figure 2: Directory and Cache controller Protocols of Migratory Scheme

cient protocols. We call the higher level protocols *atomic transaction* protocols and the lower level protocols *split transaction* protocols. In all the approaches we have seen, it is assumed that protocol designers must take charge of creating the split transaction protocols manually for obtaining maximally efficient implementations. However, this approach is highly error-prone. The main difficulty is the *huge* semantic gap between shared memory models and split transaction protocols. Protocol synthesis is feasible because the techniques employed by designers for handling ownership transfers, performing invalidations, and speculative protocol execution in real protocols tend to recur quite frequently. Protocol synthesis will allow verification to be employed at a higher level of protocol representation. This, in addition to reducing the state space, will render DSM protocol behavior *semantically closer to human intuition*, unlike in today's approaches where the low level state-space mixes the important (e.g., synchronization completing) with the mundane (how one message overtakes another inside a buffer).

Previous work in our group [30] has demonstrated the feasibility of protocol refinement. We defined a formal refinement relation, and showed, using the theorem prover PVS [32], that the protocols synthesized by our rules stand in this relation with respect to the atomic transaction protocols. It was shown there that the atomic transaction protocol for a two processor Avalanche system [13] had only 54 states, whereas the split transaction protocol had 23,000 states. Since our aim is to do the verification on the atomic transaction protocol, this has the potential to significantly reduce the state space. We briefly summarize our past work in the next section.

3.3.1 A brief overview of protocol synthesis through refinement

We adopt the CSP notation of Hoare [21] and model ownership transfers as *atomic* transaction steps (using the rendezvous construct of CSP). In Figure 2, we illustrate such a protocol description. In this description, the ownership of a cache line is managed under a *migratory-style* protocol: the line access rights migrate from node to node. As a specific example, assume that there are two cache controllers C1 and C2 executing a cache node protocol beginning in state `I` (invalid), and one directory controller D executing the directory protocol beginning in state `F` (the line is 'free'). When C1 suffers a cache miss (the `rw` transition), it attains state `T1`, and sends a request to the directory controller. We show the act of sending this request through the rendezvous statement `h!req`. This statement is jointly executed with the matching statement `r(i)?req` of D. Note that we do not mention *how* this rendezvous is realized. All that the specification writer needs to think about is that the state of D and C1 (viewed as a pair) advances from `(F,T1)` to `(T1,T2)` *atomically*. D then executes the `r(j)!gr(data)`, granting the data and line ownership to

C1. This again causes an atomic advancement of the state from `(T1,T2)` to `(E,V)`. Suppose C2 now suffers a miss, attains state `T1`, and sends a request to D. The joint moves of D and C2 will now be from `(E,T1)` to `(I1,T2)`, the state where D is preparing to invalidate C1 (which is, recall, in state `V`). Then, D and C1 execute the joint move `(I1,V)` to `(I2,T4)` to `(I3,I)`, completing the invalidation of C1. Then, D and C2 execute the joint move `(I3,T2)` to `(E,V)`, where D now records the ownership of C2.

Notice that while D and C2 were in state `(I1,T2)`, C1 could have decided to autonomously evict the cache line, attaining state `T3`. Unfortunately, if D now initiates the rendezvous `r(o)!inv`, instead of a matching action occurring, what occurs is a rendezvous initiation by C1 for another statement, namely `h!LR(data)`. This is quite akin to situations that have been faced by past researchers when they studied the problem of implementing CSP on a distributed platform using *generalized input- and output guards* [12]. Unfortunately, their solutions are too "heavy weight" for our purposes. Our solution to handle the above "apparent deadlock" situation is to somehow make D aware that it has been *nack*ed (in our static priority scheme, it is D that always backs-off in such situations). We then make D bounce back to state I1, and participate in the `LR` rendezvous which can now succeed.

Synthesis into split transactions

Split transactions (handshakes) consist of asynchronous (non-blocking) *send*s and receives. A rendezvous construct `h!req` is initiated through an asynchronous send, denoted `h!!req`. The handshake completes when a reply message `h??req` is received back. For simplicity, in our past work we disallowed a cache controller process from having a generalized guard (input and output actions present). In fact, if a cache controller process has an active rendezvous (of the kind '`p!E`'), then it must be the only guard in a communication state[3]. Thus, the rendezvous of a cache controller must not fail - while that of the directory controller can be *nack*ed. (This decision stems from what is really practical: a directory controller can be *nack*ed; however, if a CPU misses on a cache line, and its cache controller sends a request for that line, there is no option but to supply that line.) We also imposed the discipline that a cache controller may not communicate directly with another cache controller. While this decision simplified our algorithm, it prevented certain advanced types of refinements to be elaborated shortly. We will revise these decisions in our future work.

A brief overview of our synthesis procedure is as follows. The cache controller is either in an 'internal' (non-communication) state or a communication state [4]. Assume that it is in the latter. It must then reserve a buffer location to receive a reply, and then send out a request for rendezvous. If this request is *nack*ed (say, because of buffer capacity running out at the directory controller), it must retry its request. The directory controller, on the other hand, must reserve *two* free locations before it initiates a rendezvous. The extra location is to prevent the 'tail of buffer livelock' scenario illustrated earlier. It must try to engage in one of the rendezvous allowed by its communication state. Unlike a cache controller, a directory controller can be written with generalized guards. It can try these guards in turn. Figure 3 illustrates the *refined* (split

[3] A communication state is one where one of the guards is a communication action.

[4] Notice that as in CSP 1978, we named processes directly and did not use channel names. We may revise this decision in the present work.

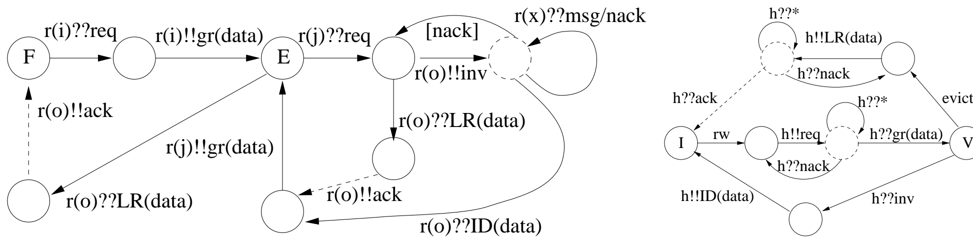

Figure 3: Refined Directory and Cache controller Migratory Protocols

transaction) protocols of the directory and cache controllers. While space precludes a full explanation, the asynchronous handshaking and details of `nack` handling are highlighted by this figure. We argued, using paper-and-pencil, that our algorithm ensures forward progress. Our algorithm can also elegantly handle the earlier mentioned 'unexpected' situation' of P2 relinquishing a line concurrently with the directory controller requesting it.

We showed, using the PVS theorem prover [33], that the split transaction protocol provides all (and only) those executions that are allowed by the atomic transaction protocol. In other words, the sequence of completed rendezvous in the implementation is one of the ones allowed by the specification, and furthermore, all the ones in the specification are realized by the implementation. *Thus, if a designer writes code to performs cache- and directory update activities around such as 'synchronization skeleton' at the atomic level,* the same activities will manifest at the split transaction level. The biggest advantage of the split transaction level is, of course, that it is *implementable*! The atomic transaction level is easier to understand and verify, but cannot be implemented directly.

An Assessment

We applied the technique described in [30] by hand to the Wildfire protocol. We found that if we had the following three synthesis rules in our algorithm, we could have synthesized all high-level aspects of the Wildfire protocol:

- The three-way handoff scenario, as described above in Section 2.1.
- A scenario that stems from three-way handoff. In this scenario, P2 must keep a copy of the cache line around, should the directory be allowed to sending "forward" requests from other nodes towards P2. This capability is, actually, necessary whenever three-way handoff is used because of the arbitrary delay between when P2 relinquishes ownership to when the directory becomes aware of that.
- A still more intricate scenario found in Wildfire. This scenario must be supported only if the designer is (as in Wildfire) extremely aggressive, and allows P2 to re-acquire as well as relinquish the very same cache line multiple times, while it is waiting for the directory to receive and acknowledge its very first relinquish! (In Wildfire, this forces P2 to keep multiple versions of the line - "one for itself" and the others for "forward requests".)

These results have encouraged us to pursue the goal of protocol synthesis further. We discuss our plans for enhancing current techniques, and other future work, in the next section.

3.4 Future Work

We plan to attack the problem of verifying the correctness of DSM protocols from two different directions. One approach will be to try and develop better techniques for verifying existing split transaction protocols. Towards this end, we are in the process of developing a partial order reduction algorithm to be implemented in Murφ. This will be a reformulation of the classical partial order reduction algorithm [17], without the notion of processes. All implementations we have seen use heuristics based on partitioning the transition relation based on processes. We plan to use a different technique to partition the transition relation. The advantage is that this technique can then be used on systems that are modeled directly as transition systems, without any clear distinction of processes. We also believe that in some cases, this may result in more partial order reductions than in the classical algorithm.

Our other angle of attack is to further develop the idea of DSM protocol synthesis, by adding rules to the synthesis algorithm that exploit re-orderings permitted by the underlying memory model. This will result in more efficient synthesized protocols, that will be competitive with hand-designed protocols. Even if the synthesized protocols are not as efficient as the latter, the big advantage will be that they can be designed quickly (and, we hope, automatically), and verified more easily, since the synthesis algorithm has to be verified only once, and then, whenever a protocol is synthesized, it will be correct by design, as long as we verify the atomic transaction protocol separately.

4 Concluding Remarks

This paper reports work in progress in our group at the University of Utah on distributed shared memory protocol verification. We report our specific efforts relating to a recently proposed challenge problem by Lamport called the Wildfire challenge [26]. We are very impressed that the bug seeded in this protocol was recently discovered by one individual simply through inspection! No other success, either using inspection or verification tools, has, hitherto been reported. Even if one were to have luck verifying this particular protocol using today's tools, the state of the art of designing and verifying such protocols is nowhere close to being routine. Our long-term goals are to understand how such protocols are created, and to base design on *refinement*, as in our group's past work reported in [30]. Ultimately, we feel, the complexity of verification must be addressed *at design time*.

References

[1] The ASCI White Computer http://www.llnl.gov/asci/.

[2] The IBM Power4 Microarchitecture http://www-1.ibm.com/servers/eserver/pseries/hardware/whitepapers/power4.html.

[3] The Sun MAJC Microarchitecture http://www.sun.com/microelectronics/MAJC/.

[4] The 3GIO Third-Generation Bus Specification http://www.pcisig.com/news_room/3gio.

[5] 2001. MPV: Workshop on Specification and Verification of Shared Memory Systems, Austin, Texas, October 31, 2001 (workshop held prior to FMCAD'2000).

[6] Martín Abadi and Leslie Lamport. The existence of refinement mappings. *Theoretical Computer Science*, 82(2):253–284, 1991.

[7] Dennis Abts, David J. Lilja, and Steve Scott. Toward complexity-effective verification: A case study of the Cray SV2 cache coherence protocol, 2000.

[8] Sarita V. Adve and Kourosh Gharachorloo. Shared memory consistency models: A tutorial. *Computer*, 29(12):66–76, December 1996.

[9] Ásgeir Þór Eiríksson. The formal design of 1m-gate ASICs. *Formal Methods in Systems Design*, 16(1), January 2000.

[10] L.A. Barroso, K. Gharachoroloo, R. McNamara, A. Nowatzyk, S. Qadeer, B. Sano, S. Smith, R. Stets, and B. Verghese. Piranha: A scalable architecture based on single-chip multiprocessing. In *27th International Symposium on Computer Architecture (ISCA)*, June 2000.

[11] J. K. Bennett, J. B. Carter, and W. Zwaenepoel. Munin: Shared memory for distributed memory multiprocessors. Technical Report COMP TR89-91, Dept. of Computer Science, Rice University, April 1989.

[12] G. N. Buckley and A. Silberschatz. An effective implementation for the generalized input-output construct of CSP. *ACM TOPLAS*, 5(2):223–235, April 1983.

[13] J. B. Carter, A. Davis, R. Kuramkote, C-C. Kuo, L. B. Stoller, and M. Swanson. Avalanche: A communication and memory architecture for scalable parallel computing. In *Proc. of the Fifth Workshop on Scalable Shared Memory Multiprocessors*, June 1995.

[14] Anne Condon and Alan J. Hu. Automatable verification of sequential consistency. In *Symposium on Parallel Algorithms and Architectures (SPAA)*, July 2001.

[15] D. L. Dill, A, J. Drexler, A. J. Hu, and C. H. Yang. Protocol verification as a hardware design aid. pages 522–525, 1992.

[16] David L. Dill, Seungjoon Park, and Andreas Nowatzyk. Formal specification of abstract memory models. In Gaetano Borriello and Carl Ebeling, editors, *Research on Integrated Systems*, pages 38–52. MIT Press, 1993.

[17] Patrice Godefroid. *Partial-order methods for the verification of concurrent systems: an approach to the state-explosion problem*, volume 1032. Springer-Verlag, New York, NY, USA, 1996.

[18] Torbjörn Grahm and Barry Clark. Soc integration of reusable basband bluetooth ip. In *Proceedings of the 2001 Design Automation Conference*, pages 256–261, 2001.

[19] Hakan Grahn. *Evaluation of Design Alternatives for a Directory-Based Cache Coherence Protocol in Shared-Memory Multiprocessors.* PhD thesis, Lund University, October 1995.

[20] S. Gupta. Stanford dash multiprocessor: the hardware and software. In *Proc. of Parallel Architectures and Languages Europe (PARLE'92)*, pages 802–805, June 1992.

[21] C. A. R. Hoare. Communicating sequential processes. *CACM*, 21(8):666–677, 1978.

[22] Alan John Hu. Techniques for efficient formal verification using binary decision diagrams. Technical Report CS-TR-95-1561, 1995.

[23] P. Keleher, S. Dwarkadas, A. L. Cox, and W. Zwaenepoel. Treadmarks: Distributed shared memory on standard workstations and operating systems. In *Proc. of the Winter 1994 USENIX Conference*, pages 115–131, January 1994.

[24] J. Kuskin, D. Ofelt, M. Heinrich, J. Heinlein, R. Simoni, K. Gharachorloo, J. Chapin, D. Nakahira, J. Baxter, M. Horowitz, A. Gupta, M. Rosenblum, and J. L. Hennessy. The stanford flash multiprocessor. In *Proc. of the 21th Annual Int'l Symp. on Computer Architecture (ISCA'94)*, pages 302–313, April 1994.

[25] L. Lamport. Specifying concurrent systems with tla, 1999.

[26] Leslie Lamport. The wildfire challenge problem. http://research.microsoft.com/users/lamport/tla/wildfire-challenge.html.

[27] Leslie Lamport. The temporal logic of actions. *ACM Transactions on Programming Languages and Systems*, 16(3):872–923, May 1994.

[28] Kenneth L. McMillan. *Symbolic Model Checking*. Kluwer Academic Press, 1993.

[29] Ratan Nalumasu, Rajnish Ghughal, Abdel Mokkedem, and Ganesh Gopalakrishnan. The 'test model-checking' approach to the verification of formal memory models of multiprocessors. In Alan J. Hu and Moshe Y. Vardi, editors, *Computer-Aided Verification, CAV '98*, volume 1427 of *Lecture Notes in Computer Science*, pages 464–476, Vancouver, BC, Canada, June/July 1998. Springer-Verlag.

[30] Ratan Nalumasu and Ganesh Gopalakrishnan. Deriving efficient cache coherence protocols through refinement. *Formal Methods in System Design*, 1998. To appear in a Special Issue on Unity. Dominique Mery and Beverley Sanders (Editors).

[31] A. Nowatzyk, G. Aybay, M. Browne, E. Kelly, D. Lee, and M. Parkin. The s3.mp scalable shared memory multiprocessor. In *Proc. of the 27th Hawaii Int'l Conf. on System Sciences (HICSS-27)*, volume I, pages 144–153, January 1994.

[32] S. Owre, J. M. Rushby, and N. Shankar. PVS: A prototype verification system. In Deepak Kapur, editor, *11th International Conference on Automated Deduction (CADE)*, volume 607 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, 1992.

[33] Sam Owre, Natarajan Shankar, and John Rushby. Pvs: A prototype verification system. In *11th International Conference on Automated Deduction (CADE), Saratoga Springs, NY*, pages 748–752, June 1992.

[34] Seungjoon Park. *Computer Assisted Analysis of Multiprocessor Memory Systems*. PhD thesis, Stanford University, jun 1996. Department of Computer Science.

[35] G. L. Peterson. Myths about the mutual exclusion problem. *Information Processing Letters*, 12(3), June 1981.

[36] Shaz Qadeer. Verifying sequential consistency on shared-memory multiprocessors by model checking. Technical report, SRC, December 2001. Research Report 176.

[37] Hemanthkumar Sivaraj. MPI port conducted in October 2001. Personal Communication.

[38] Brian R. Smith. Breaking the I/O Bottleneck. http://www.performancecomputing.com/features/0000side.shtml.

[39] David L. Weaver and Tom Germond. *The SPARC Architecture Manual – Version 9*. P T R Prentice-Hall, Englewood Cliffs, NJ 07632, USA, 1994.